Mlang Programming Language

Design and Specification Document

Team 16

Ashutosh Kumbhar

Girish Nalawade

Mitesh Parab

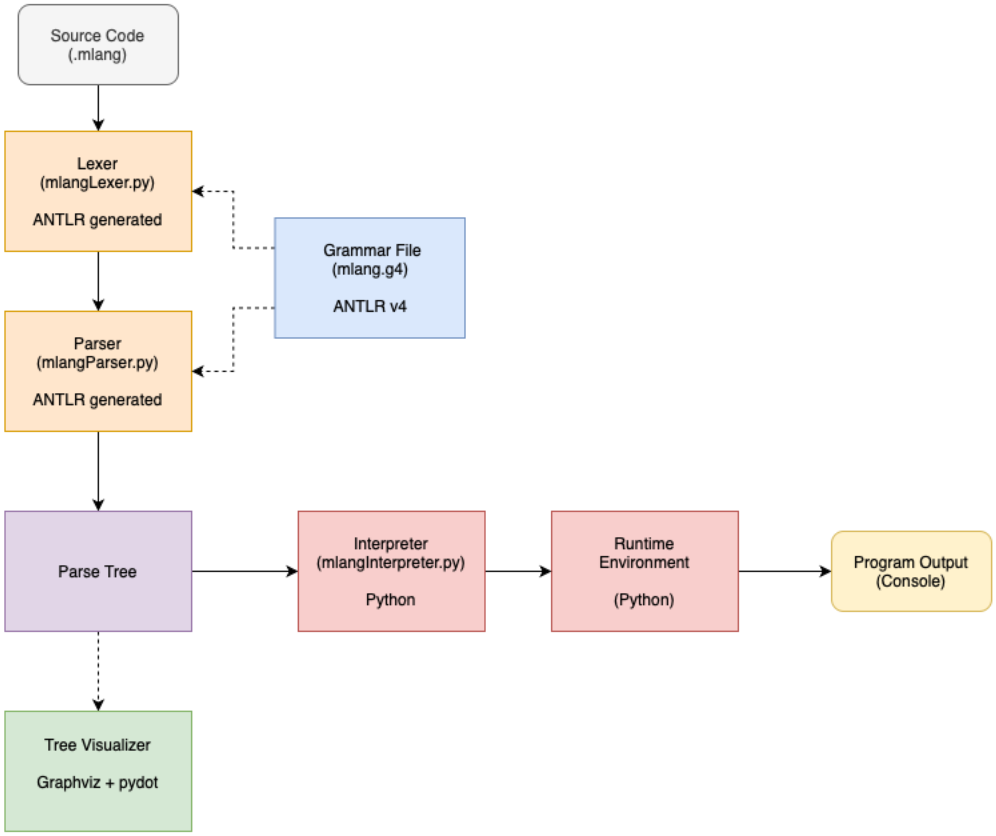
Rajesh Sawant

1. Language Overview

Name: Mlang
File Extension: .mlang
Paradigm: Imperative (statement-based control flow)
Typing Discipline: Dynamically typed (no static or strong typing)
Execution Model: Interpreted (tree-walking interpreter in Python)
Repository: https://github.com/ashuvenom/m-lang

Mlang is a domain-specific interpreted programming language designed as a cinematic and minimal storytelling language with programmable semantics. It supports variable declarations, arithmetic expressions, conditional control structures, loops, functions, lambda expressions, arrays, and more. The language uses natural language-inspired syntax and is implemented using ANTLR (for parsing) and Python (for interpretation). It uses dynamic typing and avoids enforcing strict types at compile-time, offering flexibility for expressive code.

2. Compiler Architecture



Workflow

The mlang language follows an interpreted execution pipeline that transforms cinematic-style source code into concrete output using ANTLR-generated parsing components and a custom Python interpreter. The entire pipeline is organized into discrete stages, each responsible for transforming the program closer to executable behavior.

2.1 Source Code (.mlang)

Users write programs in mlang using natural-language inspired constructs such as cast (for variable declaration), say (for printing), and cinematic control structures such as cutIf and montage. These programs are saved as plain text files with the .mlang extension, typically stored in the data/ directory. This file forms the initial input to the compilation pipeline.

2.2 Grammar Definition (mlang.g4)

mlang.g4 serves as the formal specification of the language's syntax. It defines both lexical and grammatical rules using ANTLR v4 syntax. Lexical rules (such as ID, INT, and keywords) define how individual tokens are recognized, while parser rules (such as variableDecl, printStmt, expr) define the language's structure and hierarchy.

The mlang.g4 file is processed using ANTLR to automatically generate the following Python modules:

- mlangLexer.py – handles lexical analysis (tokenization)
- mlangParser.py – builds the parse tree from tokens
- mlangVisitor.py – enables custom traversal of the parse tree

This ensures consistency and guarantees that the lexer and parser align with the same grammar.

2.3 Lexical Analysis (mlangLexer.py)

The lexer processes the raw source code and converts it into a structured stream of tokens. These tokens represent the smallest meaningful units of the language — such as identifiers, literals, operators, and control keywords.

The lexer is entirely driven by the token definitions in mlang.g4 and ensures that the source adheres to the expected vocabulary.

2.4 Parsing (mlangParser.py)

The parser consumes the token stream and constructs a hierarchical parse tree based on mlang's grammar rules. Each rule in the grammar corresponds to a node in the tree. This tree captures both the structure and intent of the source code in a form suitable for interpretation.

This structure is critical for downstream execution, as it preserves the order and nesting of operations, control flow, and scoping.

2.5 Parse Tree Representation

The resulting parse tree provides an internal structured representation of the program, where every syntactic construct is a node. This tree can be visualized using the built-in print mechanism or rendered to a PNG using pydot and Graphviz. This aids in debugging, grammar validation, and pedagogical demonstration.

The Visual Output (tree.png) is saved in data/trees/.

2.6 Interpreter (mlangInterpreter.py)

The interpreter is a Python class that implements the visitor pattern over the parse tree. It recursively visits each node, evaluates expressions, performs computations, manages variable state, and handles control flow logic.

It supports a wide variety of constructs:

- Arithmetic: +, -, *, /, %
- Booleans: truth, lie, andAlso, orElse, not
- Control flow: cutIf (if),altCut(else if), plotTwist (else), rollWhile (while), montage (for)
- Functions: scene, wrap, call
- Ternary: ..cut?... plotTwist
- Arrays and for-each loops (planned)
- Lambda expressions (planned)

The interpreter also performs runtime error checking, such as undefined variables, division by zero, and type mismatches.

2.7 Runtime Environment

The interpreter maintains an in-memory environment implemented as a Python dictionary. This acts as a symbol table for variable names, function references, array contents, and more. It is dynamically updated as the interpreter executes assignments, loops, and function calls.

The environment supports dynamic typing, which aligns with mlang's interpreted and flexible nature.

2.8 Console Output

The output of a program is visible through say statements, which print the result of evaluated expressions to the console. This is the final stage in the pipeline and represents the user's interaction with the interpreter.

In the future, this output mechanism could be extended to support logging, return values, or external I/O.

3. Language Design and features

3.1 Assignment & Variables

Construct	Description	Example
Variable Declaration	Declare a variable	cast hero is 5;
Assignment	Assign/update a variable	hero is hero + 1;

3.2 Arithmetic Operations

Operator	Meaning	Example
+	Addition (also supports string concatenation)	say 5 + 3;
-	Subtraction	say 10 - 2;
*****	Multiplication	say 2 * 4;
/	Integer Divison	say 8 / 2;
%	Modulo	say 5%3;

3.2 Control Flow Statements

Keyword	Meaning	Example
cutIf	if	cutIf hero sameAs villain action say “Twist!”; cut
altCut	Else if	cut altCut villain biggerThan 5 action say “Backup plan!”; cut
plotTwist	else	plotTwist action say “Unexpected move.”; cut
rollWhile	While Loop	rollWhile x smallerThan 10 action ... cut
montage	For Loop	montage cast i is 0; i smallerThan 5; i is i + 1 action say i; cut
pause	Break loop (Optional)	montage ... action pause; cut
replay	Continue (Optional)	montage ... action replay; cut

3.4 Boolean Operations

Keyword	Meaning	Example
truth	Boolean true	cast spotlight is truth;
lie	Boolean false	cast rumor is lie;
sameAs	Equals	cutIf hero sameAs villain action say “What a twist!”; cut
notSame	Not equals	cutIf hero notSame villain action say “A rivalry unfolds.”; cut
biggerThan	Greater than	cutIf boxOffice biggerThan 100 action say “Blockbuster hit!”; cut

biggerOrEqual	Greater than or equal	cutIf rating biggerOrEqual 75 action say "Certified Fresh!"; cut
smallerThan	Less than	cutIf rating smallerThan 5 action say “Critics hated it.”; cut
smallerOrEqual	Less than or equal	cutIf budget smallerOrEqual 10 action say "Indie vibe!";cut
andAlso	Logical AND	cutIf truth andAlso spotlight action say “Rolling camera!”; cut
orElse	Logical OR	cutIf storm orElse sabotage action say “Scene delayed.”; cut

3.5 Data Types (Dynamically Typed)

All values in mlang are dynamically typed — there are no explicit type declarations. Type is inferred from the assigned value

Type	Description	Syntax Example	Equivalent in Python
Integer	Whole numbers	cast score is 10;	int
Boolean	truth / lie	cast flag is truth;	bool
String	Sequence of characters	cast name is "Raj";	str
Array	List of values	cast items is [1, 2, 3];	list
Function	User-defined via scene/wrap	scene greet with name action ... cut	def greet(name): ...

Note: Unlike other types, truth and lie are reserved keywords representing boolean values.

3.6 Function Constructs

Functions in mlang are defined using cinematic metaphors. A function is declared using the keyword scene, parameters are specified with with, return values are specified using wrap, and functions are invoked using call.

Keyword	Purpose	Example	Equivalent in Python
scene	Declare a function	scene greet with name action ... cut	def greet(name): ...
with	Specify function parameters	scene add with a, b action ... cut	def add(a, b): ...
wrap	Return from a function	wrap result;	return result

call	Call a function	cast x is call add with 2, 3;	x = add(2, 3)

3.7 Ternary Operator

mlang supports Java-style ternary expressions using the cinematic keywords cut? and plotTwist. These allow condition-based inline expressions — ideal for concise, readable decisions.

Syntax

<condition> cut? <true_expr> plotTwist <false_expr>

- If the condition evaluates to truth → returns true_expr
- Otherwise → returns false_expr

Example

cast verdict is rating biggerThan 90 cut? "Oscar" plotTwist "Snubbed";

say verdict;

Equivalent in Java:

String verdict = (rating > 90) ? "Oscar" : "Snubbed";

3.8 Keyword Dictionary

Keyword	Meaning	Equivalent (Traditional)
cast	Variable declaration	var / let
is	Assignment	=
say	Output to console	print(...)
note:	Single-line comment	// or #
cutIf	If condition	if
altCut	Else-if condition	else if
plotTwist	Else clause	else
rollWhile	While loop	while
montage	For loop	for (...)
pause	Break from loop	break

replay	Continue loop	continue
scene	Function declaration	def / function
with	Function parameters	(a, b)
wrap	Return from function	return
call	Function call	foo(a, b)
action	Begin code block	{
cut	End code block	}

3.9 Reserved Keywords

These cinematic keywords are reserved and cannot be used as identifiers for variables, functions, or parameters.

Category	Reserved Keywords
Language Syntax	cast, is, say, note:, action, cut
Control Flow	cutIf, altCut, plotTwist, rollWhile, montage, pause, replay, cut?
Functionality	scene, with, wrap, call
Boolean Literals	truth, lie
Comparison Ops	sameAs, notSame, biggerThan, smallerThan
Logic Operators	andAlso, orElse, not

4. Grammar Specification

mlang adopts cinematic storytelling elements as syntactic constructs — such as using scene for function definitions, cutIf and plotTwist for control flow, and wrap for return statements — to create a visually and semantically expressive language.

4.1 EBNF Grammar

The following Extended Backus–Naur Form (EBNF) defines the core structure of mlang programs:

```
program         ::= { statement } EOF ;

statement       ::= variableDecl
                  | assignStmt
                  | printStmt
                  | ifStmt
```

```
                  | whileStmt
                  | forStmt
                  | breakStmt
                  | continueStmt
                  | functionDecl
                  | functionCall
                  | returnStmt ;

variableDecl    ::= "cast" ID "is" expr ";" ;
assignStmt      ::= ID "is" expr ";" ;
printStmt       ::= "say" expr ";" ;

ifStmt          ::= "cutIf" expr "action" { statement } "cut"
                    { elifStmt } [ elseStmt ] ;

elifStmt        ::= "altCut" expr "action" { statement } "cut" ;
elseStmt        ::= "plotTwist" "action" { statement } "cut" ;

whileStmt       ::= "rollWhile" expr "action" { statement } "cut" ;

forStmt         ::= "montage" variableDecl expr ";" assignStmt
                    "action" { statement } "cut" ;

breakStmt       ::= "pause" ";" ;
continueStmt    ::= "replay" ";" ;

functionDecl    ::= "scene" ID [ "with" paramList ] "action"
                    { statement } "cut" ;

functionCall    ::= functionCallExpr ";" ;
functionCallExpr::= "call" ID [ "with" argList ] ;

returnStmt      ::= "wrap" expr ";" ;

paramList       ::= ID { "," ID } ;
argList         ::= expr { "," expr } ;

expr            ::= expr "cut?" expr "plotTwist" expr
                  | expr "*" expr
                  | expr "/" expr
                  | expr "%" expr
                  | expr "+" expr
                  | expr "-" expr
                  | expr "andAlso" expr
                  | expr "orElse" expr
                  | expr "sameAs" expr
                  | expr "notSame" expr
                  | expr "smallerThan" expr
```

```
                  | expr "biggerThan" expr
                  | expr "biggerOrEqual" expr
                  | expr "smallerOrEqual" expr
                  | "not" expr
                  | functionCallExpr
                  | BOOL
                  | INT
                  | STRING
                  | ID ;

BOOL            ::= "truth" | "lie" ;
ID              ::= (letter | "_") { letter | digit | "_" } ;
INT             ::= digit { digit } ;
STRING          ::= '"' { any-character-except-quote-or-newline } '"' ;
COMMENT         ::= "note:" { any-character-except-newline } ;
```

4.2 ANTLR Grammar (mlang.g4) (updated till milestone 2)

The following ANTLR grammar is used to generate the lexer and parser for mlang using ANTLR 4:

```
grammar mlang;

program: statement* EOF;

statement
    : variableDecl
    | assignStmt
    | printStmt
    | ifStmt
    | whileStmt
    | forStmt
    | breakStmt
    | continueStmt
    | functionDecl
    | functionCall
    | returnStmt
    ;

variableDecl: 'cast' ID 'is' expr ';';
assignStmt: ID 'is' expr ';';
printStmt: 'say' expr ';';

ifStmt
    : 'cutIf' expr 'action' statement* 'cut' (elifStmt)* (elseStmt)?
```

```
    ;

elifStmt
    : 'altCut' expr 'action' statement* 'cut'
    ;

elseStmt
    : 'plotTwist' 'action' statement* 'cut'
    ;

whileStmt
    : 'rollWhile' expr 'action' statement* 'cut'
    ;

forStmt
    : 'montage' variableDecl expr ';' assignStmt 'action' statement*
'cut'
    ;

breakStmt: 'pause' ';';
continueStmt: 'replay' ';';

functionDecl
    : 'scene' ID ('with' paramList)? 'action' statement* 'cut'
    ;

functionCall
    : functionCallExpr ';'
    ;

functionCallExpr
    : 'call' ID ('with' argList)?
    ;

returnStmt
    : 'wrap' expr ';'
    ;

paramList: ID (',' ID)*;
argList: expr (',' expr)*;

expr
    : expr op=('*'|'/'|'%') expr
    | expr op=('+'|'-') expr
    | expr op=('andAlso'|'orElse') expr
    | expr
op=('sameAs'|'notSame'|'smallerThan'|'biggerThan'|'biggerOrEqual'|'sma
llerOrEqual') expr
    | expr 'cut?' expr 'plotTwist' expr
    | 'not' expr
```

```
    | functionCallExpr
    | BOOL
    | INT
    | STRING
    | ID
    ;

BOOL: 'truth' | 'lie';
ID: [a-zA-Z_][a-zA-Z0-9_]*;
INT: [0-9]+;
STRING: '"' (~["\r\n])* '"';
WS: [ \t\r\n]+ -> skip;
COMMENT: 'note:' ~[\r\n]* -> skip;    | BOOL
    | INT
    | STRING
    | ID
    ;

BOOL: 'truth' | 'lie';
ID: [a-zA-Z_][a-zA-Z0-9_]*;
INT: [0-9]+;
STRING: '"' (~["\r\n])* '"';
WS: [ \t\r\n]+ -> skip;
COMMENT: 'note:' ~[\r\n]* -> skip;
```

The above grammar has been implemented in ANTLR (v4.13) and used to generate the lexer, parser, and visitor files. All constructs have been tested with working parse trees and a functioning interpreter

5. Sample Programs

5.1 matrixChoice.mlang (*cutIf, altCut, plotTwist)*

```
note: The Oracle presents the choice

cast pill is "red";

cutIf pill sameAs "red"
action
  say "You chose to know the truth.";
```

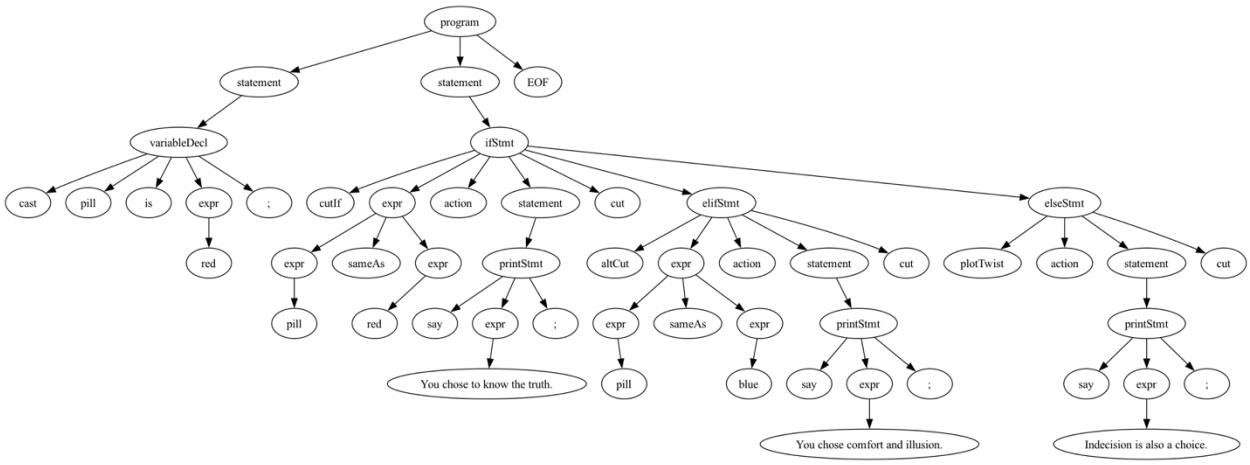
```
cut
altCut pill sameAs "blue"
action
  say "You chose comfort and illusion.";
cut plotTwist
action
  say "Indecision is also a choice.";
cut
```

Output:

```
You chose to know the truth.
```

Parse tree:

```
 program (statement (variableDecl cast pill is (expr "red") ;))
(statement (ifStmt cutIf (expr (expr pill) sameAs (expr "red")) action
(statement (printStmt say (expr "You chose to know the truth.") ;))
cut (elifStmt altCut (expr (expr pill) sameAs (expr "blue")) action
(statement (printStmt say (expr "You chose comfort and illusion.") ;))
cut) (elseStmt plotTwist action (statement (printStmt say (expr
"Indecision is also a choice.") ;)) cut))) <EOF>)
```



5.2 bossFight.mlang *(loop + logic + ternary)*

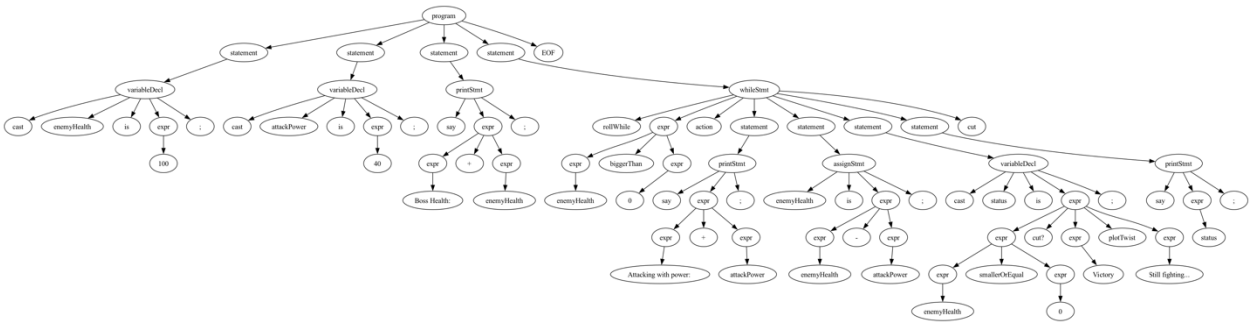
```
cast enemyHealth is 100;
cast attackPower is 40;

say "Boss Health:" + enemyHealth;

rollWhile enemyHealth biggerThan 0 action
  say "Attacking with power: " + attackPower;
  enemyHealth is enemyHealth - attackPower;

  cast status is enemyHealth smallerOrEqual 0 cut? "Victory" plotTwist
"Still fighting...";
  say status;
cut
```

Output:

```
Boss Health:100
Attacking with power: 40
Still fighting...
Attacking with power: 40
Still fighting...
Attacking with power: 40
Victory
```

Parse Tree:

```
(program (statement (variableDecl cast enemyHealth is (expr 100) ;))
(statement (variableDecl cast attackPower is (expr 40) ;)) (statement
(printStmt say (expr (expr "Boss Health:") + (expr enemyHealth)) ;))
(statement (whileStmt rollWhile (expr (expr enemyHealth) biggerThan
(expr 0)) action (statement (printStmt say (expr (expr "Attacking with
power: ") + (expr attackPower)) ;)) (statement (assignStmt enemyHealth
is (expr (expr enemyHealth) - (expr attackPower)) ;)) (statement
(variableDecl cast status is (expr (expr (expr enemyHealth)
smallerOrEqual (expr 0)) cut? (expr "Victory") plotTwist (expr "Still
fighting...")) ;)) (statement (printStmt say (expr status) ;)) cut))
<EOF>)
```



5.3 avengersAssemble.mlang *(function + loop)*

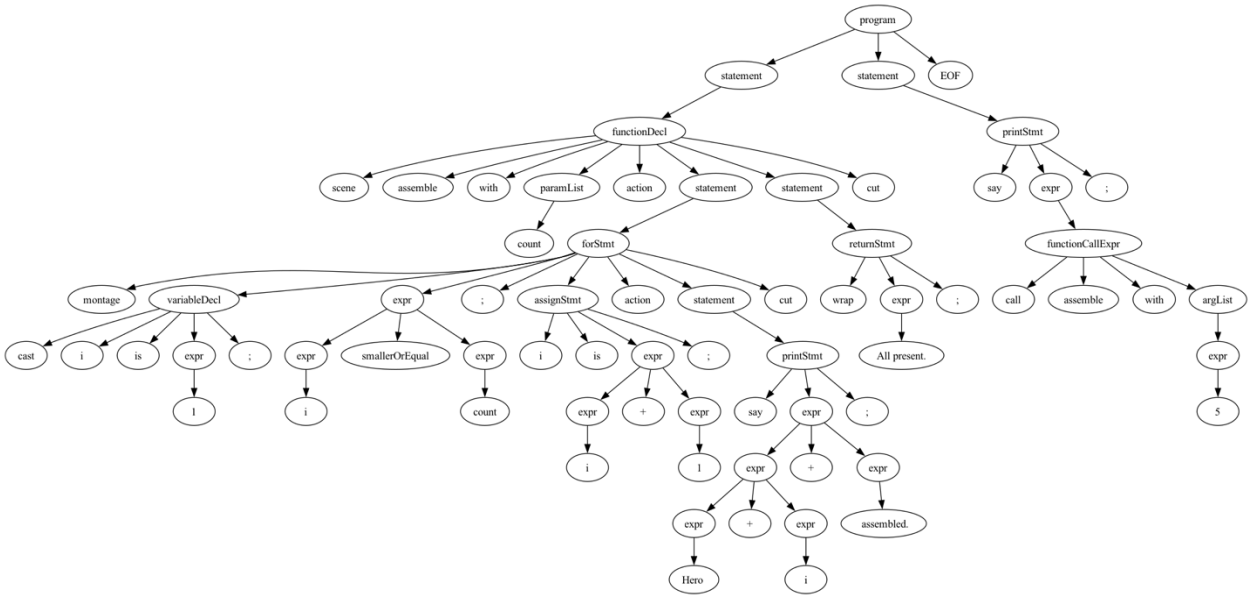
```
note: Assembling the Avengers

scene assemble with count action
  montage cast i is 1; i smallerOrEqual count; i is i + 1;
  action
    say "Hero " + i + " assembled.";
  cut
  wrap "All present.";
cut

say call assemble with 5;
```

Output:

```
Hero 1 assembled.
Hero 2 assembled.
Hero 3 assembled.
Hero 4 assembled.
Hero 5 assembled.
All present.
```

Parse Tree:

```
(program (statement (functionDecl scene assemble with (paramList
count) action (statement (forStmt montage (variableDecl cast i is
(expr 1) ;) (expr (expr i) smallerOrEqual (expr count)) ; (assignStmt
i is (expr (expr i) + (expr 1)) ;) action (statement (printStmt say
(expr (expr (expr "Hero ") + (expr i)) + (expr " assembled.")) ;))
cut)) (statement (returnStmt wrap (expr "All present.") ;)) cut))
(statement (printStmt say (expr (functionCallExpr call assemble with
(argList (expr 5)))) ;)) <EOF>)
```


program
statement
statement
EOF
functionDecl
printStmt
scene
assemble
with
paramList
action
statement
statement
cut
say
expr
;
count
forStmt
returnStmt
functionCallExpr
montage
variableDecl
expr
;
assignStmt
action
statement
cut
wrap
expr
;
call
assemble
with
argList
cast
i
is
expr
;
expr
smallerOrEqual
expr
i
is
expr
;
printStmt
All present.
expr
1
i
count
expr
+
expr
say
expr
;
5
i
1
expr
+
expr
expr
+
expr
assembled.
Hero
i